LISP as the language for an incremental computer

by

Lionello A. Lombardi* and Bertram Raphael**

March 1964

LISP as the language for an incremental computer

by

Lionello A. Lombardi* and Bertram Raphael**

March 1964 51-64

Work reported herein was partly supported by Project MAC, an M. I. T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

* Associate Professor of Industrial Management, M. I. T.

** Research Assistant, Mathematics Department, M. I. T.

1. General

The following two characteristics are commonly found in information system for the command and control of complex, diversified military systems, for the supply of information input for quantitative analysis and managerial decision making, and for the complementation of computer and scientist in creative thinking ("synnoesis") [10].

1) the input and output information flows from and to a large, continuous, on-going, evolutionary data base;

2) the algorithms of the process undergo permanent evolution along lines which cannot be predicted in advance.

Most present day information systems are designed along ideas proposed by Turing and von Neumann. The intent of those authors was to automate the execution of procedures, once the procedures were completely determined. Their basic contributions were the concepts of "executable instructions", "program" and "stored program computer". Information systems based on this conventional philosophy of computation handle effectively only an information process which 1) is "self-contained", in the sense that its data have a completely predetermined structure, and 2) can be reduced to an algorithm in final" form, after which no changes can be accomodated but those for which provision was made in advance. Consequently, the current role of automatic information systems in defense, business and research is mainly confined to simple routine functions such as data reduction, accounting,and lengthy arithmetic computations. Such systems cannot act as evolutionary extensions of human minds in complex, changing environments.

List-processing computer languages [7] have introduced a flexible and dynamically-changable computer memory organization. While this feature permits

the manipulation of new classes of data, it does not solve the basic communication problems of an evolutionary system. Each program must still "know" the form of its data; and before any processing takes place, a complete data set containing a predetermined amount of data must be supplied.

Multiple-access, time-shared, interactive computers [8] cannot completely make up for the inadequacies of conventional and list-processing systems. With time-sharing, changes in systems being developed can be made only by interrupting working programs, altering them, and then resuming computation; no evolutionary characteristics are inherent in the underlying system of a multiple-access, time-shared computer. Thus, as preliminary usage confirms, multiple-access time sharing of conventional computers is useful mainly in facilitating debugging of programs. While such physical means for close man-computer interaction are necessary for progress in information systems, they are not sufficient alone to produce any substancial expansion in the type of continuous, evolutionary, automatic computer service with which this paper is concerned.

2. The problem

A new basic philosophy is under development for designing automatic information systems to deal with information processes taking place in a changing, evolutionary environment. [1,5,6]. This new apporach requires departing from the ideas of Turing and von Neumann. Now the problem is not "executing determined procedures", but rather "determining procedures". Open-endedness, which was virtually absent from the Turing-von Neumann machine concept, must lie in the very foundations of the new philosophy.

The basis of the new approach is an "incremental computer" which, instead of executing frozen commands, evaluates expressions under the control of the

available information context. Such evaluation mainly consists of replacing blanks (or unknowns) with data, and performing arithmetic or relational reductions. The key requirements for the incremental computer are:

1) The extent to which an expression is evaluated is controlled by the currently available information context. The result of the evaluation is a new expression, open to accommodate new increments of pertinent information by simply evaluating it again within a new information context.

2) Algorithms, data and the operation of the computer itself are all represented by 'expressions' of the same kind. Since the form of implementation of an expression which describes an evaluation procedure is irrelevant, decision of hardware vs. software can be made case by case.

3) The common language used in designing machines, writing programs, and encoding data is directly understandable by untrained humans.

While the Turing-von Neumann computer is computation-oriented, the incremental computer is interface-oriented. Its main function is to catalyze the open-ended growth of information structures along unpredictable guidelines. Its main operation is an incremental data assimilation from a variable environment composed of information from humans and/or other processors. (Still, the incremental computer is a universal Turing machine, and can perform arithmetic computations quite efficiently).

Current research on the incremental computer is aimed at designing it with enough ingenuity to make the new principles as fruitful as the ones of Turing and von Neumann (see [1] and [5]). Some of the main study areas are: the design of the language; the class of external recursive functions and a mechanism called a discharge stack [2] for their fast evaluation; the design of a suitable memory

and memory addressing scheme (the latter problem is being attacked by means of higher order association lists); saving on transfers of information in memory and the use of cyclic lists; avoidance or repetition of identical strings within different expressions through the use of shorthands, and related problems of maintenance of free storage.

The following will present a quite elementary, restricted and perhaps inefficient version of the incremental computer based on LISP. LISP is the currently available computer language which most closely satisfies the requirements of an incremental computer system. The purpose of this presentation is to demonstrate some of the concepts of incremental data assimilation to scientists who are familiar with LISP. Features of a preliminary LISP implementation can be used as a guide in the development of the ultimate language for the incremental computer.

3. Aspects of the proposed solution

Various structures have been propsed for the language of the incremental computer, mainly stressing closeness to natural language (for preliminary studies see [3] and [4]). Here, however, we will consider the case in which this language is patterned on LISP. In this case a simplified version of the incremental computer will be represented by an extension of the normal LISP "EVALQUOTE" operator. This operator, itself programmed in LISP , will evaluate LISP expressions in a manner consistent with the principles of the incremental computer which are presented below. The LISP representations and programs for implementing these principles will be discussed in section 4 of this paper. The LISP meta-language will be used for all examples in the following sections.

i) Omitted arguments:

Suppose func is defined to be a function of m arguments. Consider the problem of evaluating

$$\text{func}[x_1;x_2; \ldots; x_n] \qquad (n < m) \qquad (1)$$

Regular LISP would be unable to assign a value to (1). However, for the incremental computer (1) has a value which is itself a function of (m-n) arguments. This latter function is obtained from (1) by replacing the appropriate n arguments in the definition of func by the specified values x_1, x_2, ..., x_n.

For example, consider the function

list 3 = λ[[x;y;z];cons[x;cons[y;cons[z;NIL]]]]

If A and (B,C) are somehow specified to correspond to the first and third arguemnts in the list 3 definition, then the incremental computer should find the value of list 3[A;(B,C)] to be

λ[[u];cons[A;cons[u;((B,C))]]]

ii) Indefinite arguments:

In regular LISP a function can be meaningfully evaluated only if each supplied argument is of the same kind -- such as S-expression, functional argument, or number -- as its corresponding variable in the definition of the function. In contrast, the incremental computer permits any argument of a function to be itself a function of further, undetermined arguments. (If these latter arguments were known, then the inner function could be evaluated before the main function, as LISP normally does.) The value of a function with such indefinite arguments should be a new function, all of whose unspecified arguments are at the top level.

For example, consider again the function list 3 defined above. In the incremental computer,

list 3 [D;λ[[u];cons[E;u]];λ[[u];car[u]]]

should evaluate to

λ[[r;s][cons[D;cons[cons[E;r];cons[car[s];NIL]]]]]

iii) Threshold conditions

Consider for example the function sum = [[x;y];x + y]. We say that the threshold condition for evaluating a sum is that both arguments of sum be supplied and that they both be numerical atoms. In general, a threshold condition is a necessary and sufficient condition for completing, in some sense, the evaluation of a function. In regular LISP, it is considered a programming error to request the evaluation of an expression involving a function whose threshold condition cannot be satisfied. In the incremental computer, on the other hand, expressions may be evaluated even though they involve indefinite or omitted arguments (as in (i) and (ii) above). In these cases the evaluation is not complete in the sense that the values are themselves functions which will require additional evaluation whenever the appropriate missing data are supplied.

Occasionally the threshold condition for a function does not require the presence of all the arguments. For example, the threshold condition associated with the logical function and is, "either all arguments are present and are truth-valued atoms, or at least one argument is present and it is the truth-valued atom representing falsity."

The incremental computer must know the threshold conditions for carrying out its various levels of evaluation. One of the most challenging problems in the theoretical design of the new incremental computer is that of determining efficient threshold conditions for arbitrary functions designed by a programmer.

The illustrative program described in the next section employs only the most obvious threshold conditions.

4. The program

Let us consider some of the problems of representation and organization which must be faced in the course of implementing a LISP version of the incremental computer.

i) Omitted arguments:

Since LISP functions are defined by means of the lambda-notation [9], the role of an argument of a function is determined solely by its relative position in the list of arguments. If an argument is omitted, the omission must not change the order number of any of the supplied arguments. This can be accomplised only if each omitted argument is replaced by some kind of marker to occupy its position. Therefore in this LISP formalism for the incremental computer each function must always be supplied the same number of arguments as appear in its definition; however, some of these arguments may be the special atomic symbol "NIL*" which indicates that the corresponding argument is not available for the current evaluation.

The evaluation of a function, some of whose arguments are NIL*'s, is approximately as follows: Each supplied argument (i.e.,each argument which is not

NIL*) is evaluated, the value substituted into the appropriate places in the definition of the function, and the corresponding variable deleted from the list of bound variables in the definition of the function. What remains is just the definition of a function of the omitted variables.

ii) Indefinite arguments:

An indefinite argument, as discussed in section 3.above, is an argument which is itself a function of new unknown arguments. The present program assumes that any argument which is a list whose first element is the atom "LAMBDA" is an indefinite argument. This convention does not cause any difficulty in the use of functional arguments, since they would be prefixed, as S-expressions, by the symbol "FUNCTION". However, there is an ambiguity between indefinite arguments and functional arguments in the meta-language. Also, it is illegal to have an actual supplied argument be a list starting with a "LAMBDA". A more sophisticated version of this program should have some unique way to identify indefinite arguments (perhaps by consing a NIL* in front of them).

The treatment of indefinite arguments is straightforward if one remembers that a main function and an indefinite argument are both λ-expressions, each consisting of a list of variables and a form containing those variables. The process of evaluating a function fn of an indefinite argument arg involves, then, identifying the variable v in the variable-list of fn which corresponds to arg; replacing v by the string of variables in the variable-list of arg; and substituting the entire form in arg for each occurrence of v in the form in fn. The treatment of a conditional function containing an indefinite argument is similar although somewhat more complicated.

iii) Conflicts of variables:

The same bound variables used in different λ-expressions which appear one within another "conflict" in the sense that they make the meaning of the over-all expression ambiguous. The use of indefinite arguments frequently leads to such conflicts. This problem is avoided in the present system by replacing every bound variable, as soon as it is encountered, by a brand new atomic symbol generated by the LISP function gensym.

iv) Threshold conditions:

Certain program simplifications can be made automatically by the incremental computer, if corresponding threshold conditions are satisfied. In particular, if every argument of a function is the symbol NIL*, then the function of those arguments is replaced by the function itself.

The incremental computer is represented by the LISP function evalquote 1. This function is similar to the normal evalquote operator except that evalquote 1 first checks to see if any incremental data processing, of the kinds discussed above, is called for. If so, evalquote 1 performs the appropriate partial" evaluations. If the given input is a normal LISP function of specified arguments, on the other hand, the effects of evalquote 1 and evalquote are identical.

Appendix I is a listing of the complete deck for a test run, and includes the definitions of evalquote 1 and all its subsidiary functions. The results of the run, showing examples of incremental data assimilation with the subst 1 function (which is identical to the normal LISP subst function), are given in Appendix II. The curious reader can understand the detailed operation of the programs by studying these listings.

5. Conclusions

We can now make the following observations concerning the use of LISP as the language for the incremental computer:

i) Although perhaps too inefficient to be a final solution, LISP is still a very useful language with which to illustrate the features of a new concept of algorithm representation. It is especially easy to use LISP to design an interpreter for a language similar to, but different in significant ways from, LISP itself.

ii) The program described in this paper is quite limited with regard to its implementation of both LISP and the incremental computer. If a more complete experimental system were desired, the present system could easily be extended in any of several directions. For example, in LISP, allowance could be made for the use of functions defined by machine-language subroutines, and the use of special forms; in the incremental computer, threshold conditions could be inserted to allow partial evaluation and simplification of conditional expressions.

iii) Replacing all bound variables by new symbols is too brutal a solution to the "conflict" problem; the resulting expressions become quite unreadable. Bound variables frequently have mnemonic significance, and therefore should not be changed unless absolutely necessary. A more sophisticated program would identify those symbols which actually caused a conflict, and then perhaps replace each offending symbol with one whose spelling is different but similar.

iv) When a function of an indefinite argument is evaluated, the form in the argument is substituted for each occurrence of a variable in the form in the function definition. Similarly, when a function has omitted arguments, those arguments which were not omitted are each evaluated and substituted for each occurrence of variables in the form in the function definition. In the interest of saving computer space, we must be sure that what is substituted is a reference to an expression, not a copy of the expression. In the interest of readability, perhaps the print-outs should similarly contain references to repeated sub-expressions, e.g. in the form of λ-expressions, rather than fully expanded expressions.

BIBLIOGRAPHY

[1] L. A. Lombardi and B. Raphael: Man-computer information systems, lecture notes of a two week course UCLA Physical Sciences Extension, July 20-30, 1964.

[2] L. A. Lombardi: Zwei Beiträge zur Morphologie und Syntax deklarativer Systemsprachen, Akten der 1962 Jahrestagung der Gesellschaft für angewandte Mathematik Mechanik (GAMM), Bonn (1962); Zeitschr. angew. Math. Mech. (42) Sonderheft, T27-T29.

[3] ———: On the Control of the Data Flow by Means of Recursive Functions, Proc. Symp. "Symoblic Languages in Data Processing", International Computation Center, Roma, Gordon and Breach, 1962, 173-186.

[4] ———: On Table Operating Algorithms, Proc. 2nd IFIP Congress, München (1962), section 14.

[5] ———: Prospettive per il calcolo automatico, Scientia (in Italian and French) Series IV (57) 2 and 3 (1963).

[6] ———: Incremental data assimilation in man-computer systems, Proc. 1st Congress of Associazione Italiana Calcolo Automatico (AICA), Bologna, May 20-22, 1963 (in press).

[7] D. G. Bobrow and B. Raphael, A Comparison of List-processing Computer Languages, Comm. ACM, expected publication April or May, 1964.

[8] M. I. T. Computation Center, The Compatible Time-Sharing Systems: A Programmer's Guide, M. I. T. Press, Cambridge, Mass., 1963.

[9] A. Church, The Calculi of Lambda-Conversion, Princeton University Press, Princeton, New Jersey, 1941.

[10] L. Fein: The computer-related science (synnoetics) at a University in the year 1975, American Scientist (49) (1961), 149-168; DATAMATION (7) 9 (1961), 34-41.

```
DEFINE ((
(EVALQUOTE1 (LAMBDA (FN X)
    (APPLY1 FN X NIL) ))

  (APPLY1 (LAMBDA (FN X A) (COND
   ((ATOM FN) (COND
      ((GET FN (QUOTE EXPR)) (APPLY1 (GET FN (QUOTE EXPR)) X A))
      ((EQ FN (QUOTE CAR)) (COND
        ((NULL* (CAR X)) (QUOTE CAR))
        ((LAM1 (CAR X)) (APP2 X (QUOTE CAR)))
        (T (CAAR X)) ))
      ((EQ FN (QUOTE CDR)) (COND
        ((NULL* (CAR X)) (QUOTE CDR))
        ((LAM1 (CAR X)) (APP2 X (QUOTE CDR)))
        (T (CDAR X)) ))
      ((EQ FN (QUOTE CONS))(COND((LAM2 X)(APP3 X A (QUOTE CONS)))
        (T (CONS (CAR X) (CADR X))) ))
      ((EQ FN (QUOTE ATOM)) (COND ((NULL* (CAR X))(QUOTE ATOM))
        ((LAM1 (CAR X)) (APP2 X (QUOTE ATOM))) (T (ATOM (CAR X))) ))
    ((EQ FN (QUOTE EQ)) (COND ((LAM2 X)(APP3 X A (QUOTE EQ)))
        (T (EQ (CAR X) (CADR X))) ))
      (T (ERROR (LIST (QUOTE APPLY1) FN X A) )) ))
    ((EQ (CAR FN) (QUOTE LAMBDA)) (APPLY2 (LAMS
        (CONS (QUOTE LAMBDA) (UNFLICT (CDR FN))) X) A))
     (T (ERROR (LIST (QUOTE APPLY1) FN X A))) ))
  (LAM1 (LAMBDA (X)
    (AND (NOT (ATOM X)) (EQ (CAR X) (QUOTE LAMBDA))) ))

  (APP2 (LAMBDA (X A)
      (LIST (CAAR X) (CADAR X) (LIST A (CADDAR X)) ))  )

  (NULL* (LAMBDA (X) (EQ X (QUOTE NIL*)) ))
  (LAM2 (LAMBDA (X) (OR
    (MEMBER (QUOTE NIL*) X) (LAM1 (CAR X)) (LAM1 (CADR X)) )))

  (APP3 (LAMBDA (X A F) ((LAMBDA (U V) (APPLY1
      (LIST (QUOTE LAMBDA) (LIST U V)(LIST F U V)) X A))
          (GENSYM) (GENSYM))))

  (LAMS (LAMBDA (FN X) (PROG (VAR1 ARG1 VARS ARGS ARG2 M L)
    (SETQ M (CADDR FN))
    (SETQ ARGS X)
    (SETQ VARS (CADR FN))
  LOOP (SETQ L (CAR ARGS))
    (COND ((LAM1 L)
           (GO FLICT)))
    (SETQ VAR1 (CONS (CAR VARS) VAR1))
    (SETQ ARG1 (CONS L ARG1))
  LOOP1 (SETQ VARS (CDR VARS))
    (COND ((NULL VARS) (RETURN (LIST (REVERSE VAR1) M
                  (REVERSE ARG1)))))
    (SETQ ARGS (CDR ARGS))
    (GO LOOP)
  FLICT (SETQ L (UNFLICT (CDR L)))
    (SETQ ARG2 (CAR L))
  LOOP2 (SETQ VAR1 (CONS (CAR ARG2) VAR1))
    (SETQ ARG1 (CONS (QUOTE NIL*) ARG1))
    (SETQ ARG2 (CDR ARG2))
    (COND (ARG2 (GO LOOP2)))
    (SETQ M (SUBST (CADR L) (CAR VARS) M))
    (GO LOOP1) )))
```

```
(UNFLICT (LAMBDA (Y ) (PROG (L )
  (SETQ L (CAR Y))
LOOP (COND ((NULL L) (RETURN Y)))
  (SETQ Y (SUBST (GENSYM) (CAR L) Y))
  (SETQ L (CDR L))
  (GO LOOP) )))

(APPLY2 (LAMBDA (L A) (COND ((MEMBER (QUOTE NIL*) (CADDR L))
        (APPLY3 L A)) (T (EVAL1 (CADR L) (PAIRLIS (CAR L)
          (CADDR L) A))) )))

(APPLY3 (LAMBDA (L A) (SEARCH (CADDR L)
  (FUNCTION (LAMBDA (J) (NOT (EQ (CAR J) (QUOTE NIL*))) ))
  (FUNCTION (LAMBDA (J) (APPLY4 L A)))
  (FUNCTION (LAMBDA (J) (LIST (QUOTE LAMBDA)(CAR L)(CADR L)))) )))

(APPLY4 (LAMBDA (L A) (PROG (VARS FORM ARGS M ARG1)
  (SETQ VARS (CAR L))
  (SETQ FORM (CADR L))
 (SETQ ARGS (CADDR L))
LOOP (SETQ ARG1 (CAR ARGS))
  (COND ((EQ ARG1 (QUOTE NIL*)) (GO B)) )
  (SETQ FORM (SUBST (LIST (QUOTE QUOTE) ARG1) (CAR VARS) FORM))
LOOP1 (SETQ ARGS (CDR ARGS))
  (COND ((NULL ARGS)(RETURN (LIST (QUOTE LAMBDA) M FORM) )))
  (SETQ VARS (CDR VARS))
  (GO LOOP)
 B (SETQ M (CONS (CAR VARS) M))
  (GO LOOP1) )))

(EVAL1 (LAMBDA (E A) (COND ((ATOM E) (COND
    ((GET E (QUOTE APVAL)) (EVAL E A))
      ((EQ E (QUOTE NIL*)) (QUOTE NIL*))(T (CDR (ASSOC E A)) ) ))
  ((ATOM (CAR E)) (COND
    ((EQ (CAR E) (QUOTE QUOTE)) (CADR E))
    ((EQ (CAR E) (QUOTE COND)) (EVCON1 (CDR E) A))
    ((EQ (CAR E) (QUOTE LAMBDA)) E)
    (T (APPLY1 (CAR E) (EVLIS1 (CDR E) A) A)) ) )
  (T (APPLY1 (CAR E) (EVLIS1 (CDR E) A) A)) ) ))

(EVCON1 (LAMBDA (C A) ((LAMBDA (X) (COND
  ((LAM1 X) (LIST (CAR X) (CADR X)
    (CONS (QUOTE COND)(CONS (LIST (CADDR X)(CADAR C)) (CDR C))) ) )
  ((EVAL1 X A) (EVAL1 (CADAR C) A))
  (T (EVCON1 (CDR C) A)) )) (CAAR C)) ))

(PAIRLIS (LAMBDA (X Y A) (COND ((NULL X) A)
    (T (CONS (CONS (CAR X)(CAR Y))(PAIRLIS (CDR X)(CDR Y) A)) )) ))

(ASSOC (LAMBDA (X A) (COND ((EQUAL (CAAR A) X) (CAR A))
    (T (ASSOC X (CDR A))) )))

(EVLIS1 (LAMBDA (M A) (COND ((NULL M) NIL)
  (T (CONS (EVAL1 (CAR M) A) (EVLIS1 (CDR M) A))) )))

( SUBST1 (LAMBDA (X Y Z) (COND ((ATOM Z) (COND
    ((EQ Z Y) X) (T Z)) )
  (T (CONS (SUBST1 X Y (CAR Z)) (SUBST1 X Y (CDR Z))))))))
  ))

EVALQUOTE1
```

```
(SUBST1  ((A  B)  C  NIL*)  )
'ALQUOTE1
(SUBST1  ((LAMBDA  (X)  (CONS  X  (QUOTE  (B))))  C  (C  Y  (C  D))))
'ALQUOTE1
(SUBST1  (NIL*  NIL*        NIL*))
'ALQUOTE1
(SUBST1  (ONION  NIL*  (LAMBDA  (X  Y)  (CONS  X  Y))  ))
'ALQUOTE1
(SUBST1  ((A  B)  C  (C  Y  (C  D))  )                )
     STOP ))))))))))
     FIN
```

```
 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 ((A B) C NIL*))


END OF EVALQUOTE, VALUE IS ..
(LAMBDA (G00003) (COND ((ATOM G00003) (COND ((EQ G00003 (QUOTE C)) (QUOTE
(QUOTE (A B)) (QUOTE C) (CAR G00003)) (SUBST1 (QUOTE (A B)) (QUOTE C) (CD


 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 ((LAMBDA (X) (CONS X (QUOTE (B)))) C (C Y (C D))))

END OF EVALQUOTE, VALUE IS ..
(LAMBDA (G00007) (COND ((ATOM (QUOTE (C Y (C D)))) (COND ((EQ (QUOTE (C
QUOTE (B)))) (T (QUOTE (C Y (C D)))))) (T (CONS (SUBST1 (CONS G00007 (QU
Y (C D))))) (SUBST1 (CONS G00007 (QUOTE (B))) (QUOTE C) (CDR (QUOTE (C Y


 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 (NIL* NIL* NIL*))

END OF EVALQUOTE, VALUE IS ..
(LAMBDA (G00008 G00009 G00010) (COND ((ATOM G00010) (COND ((EQ G00010 G0
(SUBST1 G00008 G00009 (CAR G00010)) (SUBST1 G00008 G00009 (CDR G00010)))


 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 (ONION NIL* (LAMBDA (X Y) (CONS X Y))))

END OF EVALQUOTE, VALUE IS ..
(LAMBDA (G00015 G00014 G00012) (COND ((ATOM (CONS G00014 G00015)) (COND
QUOTE ONION)) (T (CONS G00014 G00015)))) (T (CONS (SUBST1 (QUOTE ONION)
(SUBST1 (QUOTE ONION) G00012 (CDR (CONS G00014 G00015)))))))
```

ARGUMENTS..

APPENDIX II: Result of computer run

```
ND ((EQ G00003 (QUOTE C)) (QUOTE (A B))) (T G00003))) (T (CONS (SUBST1
BST1 (QUOTE (A B)) (QUOTE C) (CDR G00003))))))
```

ARGUMENTS..

```
) C (C Y (C D))))
```

```
(C D)))) (COND ((EQ (QUOTE (C Y (C D))) (QUOTE C)) (CONS G00007 (
(CONS (SUBST1 (CONS G00007 (QUOTE (B))) (QUOTE C) (CAR (QUOTE (C
B))) (QUOTE C) (CDR (QUOTE (C Y (C D)))))))))
```

ARGUMENTS..

```
OM G00010) (COND ((EQ G00010 G00009) G00008) (T G00010))) (T (CONS
T1 G00008 G00009 (CDR G00010))))))
```

ARGUMENTS..

```
Y))))
```

```
OM (CONS G00014 G00015)) (COND ((EQ (CONS G00014 G00015) G00012) (
(T (CONS (SUBST1 (QUOTE ONION) G00012 (CAR (CONS G00014 G00015)))
00014 G00015)))))))
```

```
 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 ((A B) C (C Y (C D))))

 GARBAGE COLLECTOR ENTERED AT  04050 OCTAL.                    FULL WORDS  14

GARBAGE COLLECTOR ENTERED AT  04050 OCTAL.                     FULL WORDS  14

END OF EVALQUOTE, VALUE IS ..
((A B) Y ((A B) D))

 THE TIME ( 0/ 0  000.0) HAS COME, THE WALRUS SAID, TO TALK OF MANY THINGS
```

```
FULL WORDS   1438 FREE    5329  PUSH DOWN DEPTH     105
FULL WORDS   1426 FREE    4856  PUSH DOWN DEPTH     304


OF MANY THINGS .....    -LEWIS CARROLL-
```